

The structure is simple enough for you to guess, so we will not list the HTML. Notice that no attempt has been made to align the fields, and that the prompts for the fields and the instructions for the text area flow into each other visually instead of seeming to be distinct items. Also, placing the Submit button, also called the 'Send Mail' button, on the right is unusual, and the Reset button (which you press if you decide you do not want to send a message) has been omitted altogether. See below for a discussion of how to lay out forms effectively.

Radio buttons

You obtain radio buttons in HTML by specifying an `INPUT` field with `TYPE=RADIO`. Both `NAME` and `VALUE` are mandatory.

Radio buttons are the exception to the rule that input fields must have unique names to identify their contents, as the way that you tell the browser that these radio buttons go together is to give them the same name. Their behavior is like a set of old-fashioned mechanical push-buttons on a radio set: when you press one, the previously pressed one pops out again. So, a set of radio buttons in a form returns just one, which is the one selected by the user. You will be relieved to hear that only the selected one's data is returned to the server.

The syntax is simple and regular. You give each button a name and supply some text for the user to read beside the button.

You **must** mark **exactly one** button as 'checked'. This button will be shown in the 'yes' state when the form is first displayed (and when the form is reset). If you mark more than one button as checked or none at all, it is an error. The browser's behavior is not defined, and it may do something unpredictable such as checking only the first button – or only the last one.

To help you avoid missing anything, it is best to arrange your radio button declarations in a column (this example, like the others, must be included within `<FORM>...</FORM>` tags, and with `SUBMIT` and `RESET` buttons too):

```
<P><B>Which is your favorite bean:</B><P>
<INPUT TYPE=radio NAME=coffee VALUE="Java">Old Brown Java
<INPUT TYPE=radio NAME=coffee VALUE="Sumatra">Sumatra Blue Lingtong
<INPUT TYPE=radio NAME=coffee VALUE="Mocha" CHECKED>Mocha
<INPUT TYPE=radio NAME=coffee VALUE="Jamaica">Jamaica Mountain
<INPUT TYPE=radio NAME=coffee VALUE="Kenya">Kenya Peaberry<P>
```

This setup creates a beautiful row of radio buttons like this:

Which is your favourite bean:

☐ Old Brown Java ☐ Sumatra Blue Lingtong ☒ Mocha ☐ Jamaica Mountain ☐ Kenya Peaberry

Notice that the **CHECKED** attribute is set for our favorite Mocha coffee: we supply it by default, which means initially, and when the **RESET** button is pressed. To repeat, you must make exactly one button the checked default.

Checkboxes

Checkboxes are more versatile than radio buttons, as you can have as many of them selected as you like. So if it makes sense to check two or more items, you use checkboxes, not radio buttons. Alternatively, you can use a multiple-choice **SELECT** menu. Each checkbox has just two states – checked or not checked. Most browsers use an X to indicate that a checkbox is checked.

Separate-item checkboxes

The syntax is virtually identical to that for radio buttons:

```
<P><B>Choose your breakfast:</B>
<P>
<INPUT TYPE=checkbox NAME="bacon"      CHECKED >Bacon
<INPUT TYPE=checkbox NAME="eggs"      CHECKED >Eggs
<INPUT TYPE=checkbox NAME="mushrooms" >Mushrooms
<INPUT TYPE=checkbox NAME="toast"      >Toast
<INPUT TYPE=checkbox NAME="juice"      >Juice
<INPUT TYPE=checkbox NAME="cereal"     >Cereal
```

Lining the choices up helps you to avoid mistakes

Your browser will initially display the above as something like this:

Choose your breakfast:

☒ Bacon ☒ Eggs ☐ Mushrooms ☐ Toast ☐ Juice ☐ Cereal

The user is then free to check or uncheck any or all of the boxes.

One important difference between radio buttons and checkboxes can be seen when you look carefully at the examples. With checkboxes, you can check off as many checkboxes as you please. In this case, we have assumed that everyone will want bacon and eggs for breakfast. With radio buttons, we have offered users to choose a variety of coffee via a radio button. Only one choice is possible.

Checkboxes for variations on a theme

In this example, all the checkboxes have unique names, so the breakfast menu consists of separate items. Another use of checkboxes is for variations on a single theme. For instance, you could offer your customers a range of ice-cream, rather than just one. People could have one scoop of each kind.

Your ice-cream checkboxes would be declared like this:

```
<FORM>
<H4>Please check each box for a scoop of that freshly-made
ice-cream:</H4>
<INPUT TYPE=checkbox NAME="ice" CHECKED>Vanilla
<INPUT TYPE=checkbox NAME="ice" CHECKED>Strawberry
<INPUT TYPE=checkbox NAME="ice">Pistachio
<INPUT TYPE=checkbox NAME="ice">Walnut
<INPUT TYPE=checkbox NAME="ice" CHECKED>Rum'n'Raisin
<P><INPUT TYPE=submit> <INPUT TYPE=reset>
</FORM>
```

The effect is like this:

Please check each box for a scoop of that freshly-made ice-cream:

☒ Vanilla
 ☒ Strawberry
 ☐ Pistachio
 ☐ Walnut
 ☒ Rum'n'Raisin

You can see that three scoops are pre-selected to encourage your customers to buy. They can uncheck them and choose just one scoop if they prefer.

The Submit button sends only the checked boxes' names and values to the server (as for radio buttons), whereas, just as you would expect, the Reset button puts the checkboxes back to their initial states – in this case, as shown in the figure.

The ice-cream submission includes the input field name ('ice') together with the name of each flavor of ice-cream checked at the time the Submit button is pressed. The data sent back therefore says, for example:

```
"ice=Vanilla+ice=Strawberry+ice=Rum'n'Raisin".
```

A program on the server unpacks all this again into its component parts, so that the consumer's order can be processed.

Menus (single- and multiple-choice)

Menus provide a preferred alternative to input fields when potential choices are known in advance. They use up less screen space than radio buttons (for single selections) or checkboxes (for multiple selections), so they are especially useful in instances in which numerous data must be gathered on a form.

Textual menus

The basic syntax is a `SELECT` element that contains several `OPTION` elements:

```
<P><B>Select the kinds of music you like:</B><P>
<SELECT MULTIPLE NAME="music">
  <OPTION SELECTED>Jazz
  <OPTION>Rock
  <OPTION>Blues
  <OPTION>Folk
  <OPTION>Gamelan
</SELECT>
```

The example shows three options that the user has selected at one time. If `MULTIPLE` is omitted from `SELECT`, then only one option can be selected; that is, the menu simulates radio behavior. Individual options can be marked `SELECTED`. Obviously, you can only mark one option as `SELECTED` if you have specified a single-choice menu.

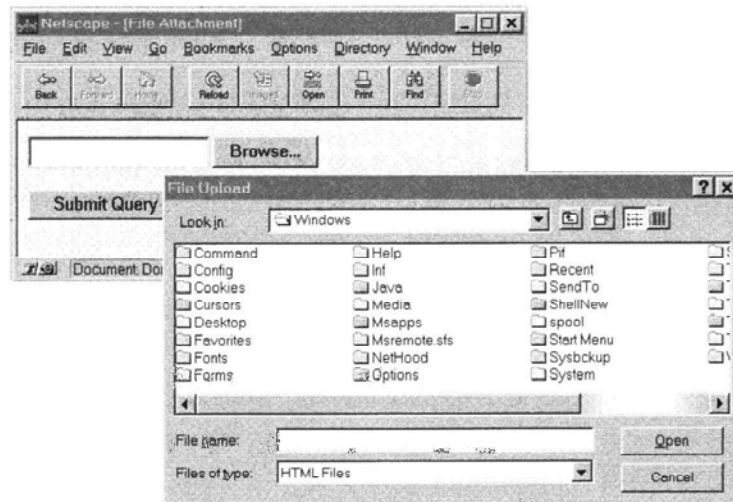
When the Submit button is pressed, the name ('music') is sent back with *each* of the selected options.

File attachments

The file attachment mechanism lets the user attach files to be submitted with the form. For instance:

```
<INPUT TYPE=file>
```

Graphic browsers normally use a field, accompanied by a button that calls up a file dialog box, to allow users to browse through their directories:



To use the file attachment, you must also put the **ENCTYPE** attribute in the FORM tag:

```
<FORM ENCTYPE="multipart/form-data"...>
```

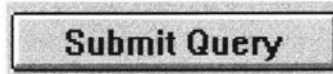
This tells the browser to be ready to submit data as files as well as files in the usual encoding.

The Submit button

The Submit button's basic structure is

```
<INPUT TYPE=submit>
```

The default button label depends on the browser. Netscape's is quite typical:



Ways you can ring the changes are:

- **VALUE** to change the default label

```
<INPUT VALUE="Submit Query Now" TYPE=submit>
```

Occasionally, this may be useful, but more likely, it can lead to confusion.

- **NAME** to distinguish which Submit button was pressed

```
<INPUT NAME="casual" VALUE="Casual Inquiry" TYPE=submit>
<INPUT NAME="urgent" VALUE="Purchase Order" TYPE=submit>
```

This makes the zipcode example look like this:

Zip or post code:

The example suggests a context in which you *might* want to provide more than one Submit button. In general, there should be exactly one – the user submits the data or not, and the server-side scripts (to handle the returned data) are as simple as possible. Prioritizing is notoriously tricky, but at least in instances in which people need to decide whether or not to spend money, they can usually get their act together and press the right button.

It is up to the server to do something sensible with the returned name and value of the Submit button.

- **SRC** to put an image on a Submit button

```
<INPUT SRC="launch.gif" ALIGN=left TYPE=submit>
```

The specified image is fetched and used instead of the default text. **ALIGN** has its usual behavior, as with **IMG** and **TABLE**; with **ALIGN=left**, for instance, the image floats down to the left margin, and allows text to wrap to its right.

If you are going to use an image Submit button, choose something visual; for example, christening an ocean liner by breaking a champagne bottle across its bows, launching a US space shuttle, or an Olympic sprinter at the starting blocks waiting for the gun.

- **TYPE=image** and **SRC=image_url** to return a location on a Submitted image

This slightly bizarre hangover from HTML 2.0 consists of an image that behaves like a Submit button, though its type is not Submit. The **x** and **y** coordinates of the click, measured from the top-left corner of the image, are returned to the server. The **NAME** attribute as usual identifies the field responsible for the input. A **VALUE** attribute can be provided (compare **ALT** for **IMG** elements) for text-only browsers: it is ignored in graphics. Netscape uses the **NAME** attribute for this purpose. An **ALIGN** attribute works exactly as for **TYPE=submit** buttons. For example:

```
<INPUT TYPE=image NAME="b" SRC="bird.gif" VALUE="clickable image">
```

The effect is to display the image as a clickable button, typically with a colored border. When the user clicks on it, x and y position data are returned URL-encoded as shown in the illustration below, in the form:

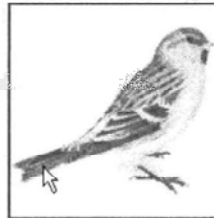
```
?name.x=xvalue&name.y=yvalue
```

For example:

```
?b.x=23&b.y=114
```

The server can use the data to determine that, in this example, the tail of the bird was clicked on, and can provide information accordingly.

```
form.html?b.x=23&b.y=114
```



- **TYPE=button**
Finally, a **BUTTON** type is provided for use with scripts such as Java. Programmers can make scripts respond to clicks using the **onClick** intrinsic event, for example to run consistency checks or to fill in derived fields (Gender must be male if Title=Mr, for instance).

The Reset button

The Reset button's basic structure is

```
<INPUT TYPE=reset>
```

The button's action is to reset all fields in the form to their initial values, which may be blank.

It has exactly the same **VALUE** and **SRC** variations as the Submit button. There is surely no reason to display more than one Reset button on any

form. We do not suggest modifying the Reset button's default appearance unless you have very good reason to do so. An inherent advantage for the default Submit and Reset buttons is that they require no image traffic between server and client. They are also highly recognizable in their standard form.

Hidden fields

If you need to preserve some status information, such as a transaction identifier, with the form data, then a hidden field is the way to do it. HTTP servers do not usually remember such information, as it would require a lot of processing and could slow down the server's response to users.

The user sees nothing special, but the contents of any hidden fields are sent with the rest of the form data. It is up to your server to do something sensible with the information; and, indeed, to ensure that something sensible was put in the field to start with. You need to discuss this with your server provider if you have any special requirements.

```
<INPUT TYPE=hidden NAME=transact VALUE=T501">
```

Password fields

You can ask users to supply a password in a form:

```
<P>Your Password : <INPUT TYPE=password NAME=pwd>
```

This creates an input field like this:

Your Password :

The password type is exactly like single-line text input, except that a star or space is echoed to the screen in place of each character of the actual password text typed in by the user. Existing HTTP sends the data in plaintext, so your users' passwords are in no way secure against sniffer programs. Secure HTTP, for example, will improve on this.

11.5 Submitting forms

The user fills in your elegant masterpiece and presses the Submit button. This sends the complete set of inputs to the URL specified by the **ACTION** attribute; it uses the method specified in the **METHOD** attribute. The two

possible methods are **get** and **post**. **get** appends the returned data to the URL, whereas **post** sends the data separately.

```
<!--call a user-written program-->
<FORM ACTION="http://www.mammon.com/programs/savezip" METHOD=post>
```

This says that the `savezip` handler program on your server is to process the encoded form data, which is posted separately. For this, you have to use HTTP.

If you do not specify a recipient with **ACTION=...** HTML simply sends the data to the document URL, which usually will not be what you want. You need to sort out with your server provider (preferably before you start designing your forms) how to deal with returned data: it is likely they have some standard software for the job. The obvious thing to do is to stick it in a file, which you fetch with FTP or which they mail to you every so often.

It is outside the scope of this book to describe how to set up a server complete with form handlers. A recent book with good, intelligible coverage of all this is *Spinning the Web: How to Provide Information on the Internet*, by Andrew Ford, International Thomson Publishing, 1996. Ford's account of how to handle data returned from forms is necessarily a bit more technical than most of our descriptions, and demands some familiarity both with programming and with the UNIX operating system.

11.6 Form design

Design principles

A great deal has been written about form design, not only for user interfaces, especially with databases, but also in bureaucracies around the world. We would not want to add to the mountains of paper on the subject. Some useful principles to bear in mind are:

- Regularly laid-out forms are easier to understand and are more likely to be completed.
- Formatting in a single column is easy and effective.
- It is simple to align prompts (using decimals) if they all end in ':' or another shared character.
- Related fields (title, forename and surname) should be together, e.g., on a line.
- Dividing lines (<HR>, table borders) and background color can visually group related fields.

Freedom of layout

HTML gives you a free choice of form layout, allowing you to use any of the mark-up elements discussed in this book. This section discusses how you can use lists and tables to organize your forms. You could take other routes: for example, forms within `<PRE>...</PRE>` tags can be laid out very simply with spaces and line breaks. The price you pay is that forms in PRE come out in fixed-pitch text, in place of the more usual, and more attractive, proportional fonts.

Platform independence

Typically, other forms languages give you a particular language intended for use on one computer platform that also comes with one, and only one, software tool, such as a database. In contrast, HTML is platform independent; it is completely portable, its browsers and servers are available on numerous machines and operating systems, and it can readily interface to nearly any database or software tool.

This 'platform independence' is a very desirable property. If you develop your forms for a Microsoft Windows NT server running on a personal computer and you find it is unable to handle all the business it generates, you can move to a fast UNIX box without changing your HTML at all. Of course, you will have to modify any scripts that depend on the operating system.

Similarly, your users can look at your forms with browsers running on UNIX workstations, Apple Macintoshes, Personal Computers, or even traditional ASCII text-only terminals. Provided their browsers correctly interpret the HTML, the exact appearance on their screens is not important. They can fill in your forms and submit them as if they were all using identical machines.

Single-column forms

Many simple form applications can be handled quite adequately using a single column of input elements, such as text fields, radio buttons, and so on. At least three ways to achieve respectable-looking, single-column forms exist:

- Using lists
- With a table using right-alignment for prompt labels and left-alignment for fields
- With a single table cell, using decimal-alignment to center the prompt characters

We will demonstrate each approach in turn.

Lists

Both plain lists (<UL PLAIN>) and definition lists (<DL>) offer possibilities for arranging input fields. Here is a simple example using definition lists:

```
<FORM>
  <DL>
    <DT>Initials:<DD><INPUT NAME="initials" SIZE="5" >
    <DT>Surname: <DD><INPUT NAME="surname" SIZE="20">
    <DT>Zipcode: <DD><INPUT NAME="zipcode" SIZE="9" >
  </DL>
  <INPUT TYPE=reset> <INPUT TYPE=submit>
</FORM>
```

In a typical browser, the result (Reset and Submit buttons not shown) is like this:

The effect is not unpleasant, but it takes up more screen space than it should, and you have no control over the positioning of the fields: in this browser, the indentation is arguably too small for our purpose. You may be able to use a style sheet to organize things better for you.

Right- and left-aligned table data cells

A more powerful, but messier approach is to use a table with two data cells (<TD>) for each row: the one on the left is to contain right-aligned text prompts; the one on the right, left-aligned fields.

```
<FORM>
  <TABLE>
    <TR> <TD ALIGN=right>Initials:
        <TD ALIGN=left ><INPUT NAME="initials" SIZE="5" >
    <TR> <TD ALIGN=right>Surname :
        <TD ALIGN=left ><INPUT NAME="surname" SIZE="20">
    <TR> <TD ALIGN=right>Zipcode :
        <TD ALIGN=left ><INPUT NAME="zipcode" SIZE="9" >
  </TABLE>
  <INPUT TYPE=reset> <INPUT TYPE=submit>
</FORM>
```

The effect is something like this:

Initials:

Surname:

Zipcode:

Two cells in table

Right-aligned prompts

Left-aligned fields

This is much more like what most database people expect from a form.

A single decimally-aligned table data cell

A more compact and arguably more elegant way to achieve the same result is to align all the fields vertically within a single table cell, by specifying a convenient character to which to align. This mechanism is designed for lining up decimal points with digits both sides, but is equally fine for aligning colon prompts with text one side and fields the other – it is a neat example of the generality of HTML. You specify 'ALIGN=char' and state that the alignment character is to be a colon (CHAR=":"). Remember to put in line breaks (
) to force browsers to format the fields in a column.

For example:

```
<FORM>
  <TABLE><TR ALIGN=char CHAR=":">
    <TD>
      Initials:<INPUT NAME="initials" SIZE="5" ><BR>
      Surname :<INPUT NAME="surname" SIZE="20"><BR>
      Zipcode :<INPUT NAME="zipcode" SIZE="9" ><BR>
    </TD>
  </TABLE>
  <INPUT TYPE=reset> <INPUT TYPE=submit>
</FORM>
```

The effect is the same.

Multi-column forms

The general case is naturally a form that is too complex to fit into the single-column mould, even if that is distorted to allow 'twin' fields (such as month and year) to share a line. The main problem is to align fields that have the appearance of regularity, although they are different sizes. Some assumption has to be made about the likely space available on the typical browser's window.

The table is the only construct in HTML that permits the required degree of precision in aligning and positioning. Many cells and rows may be needed, and with careful design, to achieve a visually satisfying result.

You can nest tables, so you can set up a large table that contains, in one or more of its data cells, a column-of-fields table, as described above. You can then save screen space by placing two or more columns beside each other. Remember, though, that this could force some users to scroll sideways to see some of the fields. It is wise to remember that many users will have screens capable of displaying lines of no more than 80 characters.

11.7 HTML 4 extensions to forms



In this section of proposed improvements for HTML 4:

- Why more powerful forms are needed
- Providing access keys
- Enabling tabbing
- Adding tooltip/balloon help information
- Adding clickable labels to controls
- Grouping fields into accessible sets

Why more powerful forms are needed

Earlier versions of HTML provided simple forms, which lacked several features that users are coming to expect. For example:

- The tabbing order for moving between fields or hypertext links was undefined.
- There was no provision for keyboard shortcuts for particular actions, nor for access keys to drive menus.
- The label text for radio buttons and checkboxes was not active; that is, clicking on label text did not push the neighboring button.
- There was no means of marking up groups of related form fields to support speech-based browsers for the print-impaired.

- There was no provision for checking values as they were entered into form fields. All checking was done at the server, when the form's contents had been submitted.
- The range of form-field types was limited in comparison with modern user interfaces. For instance, there were no data entry grids or number entry sliders, while forms themselves were restricted to single pages.
- Servers could not update individual fields in submitted forms; instead, they had to send complete HTML documents back to clients, causing screens to flicker as browsers hurried to redraw entire displays.

It is proposed to introduce several elements and attributes into HTML 4 to remedy some of these problems. They should make forms easier and quicker to use. The newcomers are:

- **ACCESSKEY**=letter
- **TABINDEX**=number *and* **NOINDEX**
- **TITLE**="little bit of help that pops up"
- **<LABEL>**
- **<FIELDSET>** *and* **CAPTION**="name of fieldset"
- Support for using **OBJECT**s as form fields

Some of these are comfortably backward-compatible with HTML 3; others cannot be, either because they need scripts or because old browsers are unable to display them sensibly. Further extensions are anticipated, for instance: better control types, enhanced menus, multiple-page layouts and server-driven updates. These exciting possibilities are not discussed further here.

Providing access keys

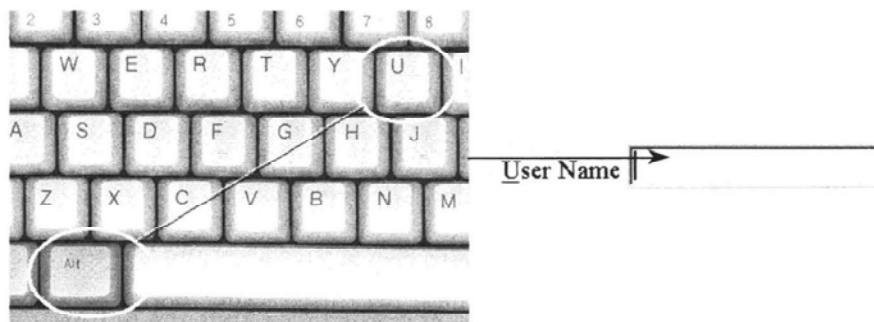
ACCESSKEY is proposed to be a single-character attribute on **LABEL**, **CAPTION** and **A** (anchor) tags. Its purpose is to make forms accessible to systems without a mouse, especially those designed for people with disabilities. It also offers quick performance for advanced users.

ACCESSKEY provides a case-insensitive keyboard shortcut for clicking on a label or anchor. So when the access key is 'U', for example, it does not matter whether the user types 'u' in upper- or lower-case characters. On Windows, the first instance of the access key character in the label or anchor is Underlined. Any single-character key can be used for access. Special keys like Enter and Control cannot be used, and it is unwise to

select keys such as £ or Ü, which do not occur on all keyboards. Each key can be used only once in any HTML page. The keyboard combination for selecting the access key is ALT (held down) and the desired character. For example:

```
<LABEL ACCESSKEY="U">User Name
  <INPUT TYPE="input" NAME="username">
</LABEL>
```

To complete this example: when the Windows user types ALT-U (prompted by the U in the field label), the input box called 'username' is given the focus, just as if the user had pointed the mouse at the field and clicked on it. (The photograph shows an English keyboard.) The presence of the focus in a text field is indicated by a vertical bar, which acts as a text cursor:



ACCESSKEY works in much the same way for hypertext anchors:

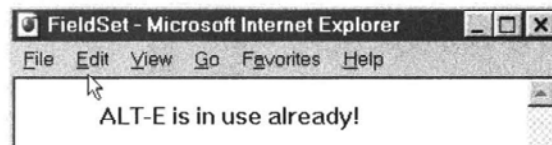
```
<A HREF="toc.html" ACCESSKEY="T">Table of Contents</A>
```

which can be rendered as shown and selected with ALT-T:

Table of Contents

Users who want to make use of access keying to activate links need to configure their browsers to highlight anchors in color, rather than by underlining.

Finally we should warn that not all legal characters are genuinely free for you to use, at least at the top level. Windows browsers typically allocate ALT-F for the File menu, ALT-E for the Edit menu, ALT-V for the View menu, and ALT-H for the Help menu, among others. You may find less room to maneuver than you expect.



Enabling tabbing

If you have several windows open on your screen, and you press the 'A' key, you might be selecting 'Acquire' if a graphics editor has the focus of attention, but the same keystroke might select 'Annotate' or 'Add' in other tools. The effect of a command depends on the current focus.

Similarly, within a document the focus can be at any of several places. It can be moved either by the mouse, or by cursor control commands (usually Tab and Reverse-Tab) from the keyboard. The 'keyboard focus' determines which object in an HTML document gets keyboard events of this sort.

Until now, most browsers have required use of the mouse to move the focus, and have ignored the keyboard altogether. This made life more difficult than it needed to be for people with disabilities. Other browsers have allowed tabbing, based on the order in which form elements, such as INPUT, appear in the HTML. The proposed HTML 4 attributes **TABINDEX** and **NOTABSTOP** identify elements, which the user can reach with the tab key.

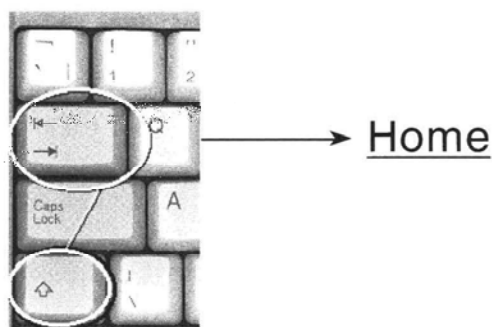
Both **TABINDEX** and **NOTABSTOP** can be supplied with any element that can accept the keyboard focus, including form fields, hypertext links, embedded objects and client-side image maps.

The TABINDEX attribute

The **TABINDEX** attribute specifies an integer value for a particular tag, forming part of the order in which the tab key traverses the document, for example:

```
<A HREF=home.html TABINDEX=4>Home Page</A>
```

If the focus was with **TABINDEX=3**, then pressing the tab key puts the focus on the Home link anchor; similarly, if the tag with **TABINDEX=5** had the focus, then a reverse tab (SHIFT-Tab) would move the focus back to our Home link again. (The photograph shows an English keyboard.)



Note that tabbing to a tag does *not* activate it. Once the user has reached the desired object (giving it the focus), whether by tabbing or pointing, it can be activated with the Enter key. If it is a hypertext link, activation means traversing the link; if it is a radio button in a form field, it means selecting that option (and deselecting the other options); if it is a checkbox it means checking or not checking that option; and so on.

The tabbing order is usually the order in which items appear in the document, unless overridden by **TABINDEX** or **NOTABSTOP**. The situation is more complicated when you have embedded objects in your HTML. Embedded objects (and objects that they contain) can form parts of the tabbing sequence. Client-side image maps can be defined to let users tab through their hotzones, for instance. Because of this complexity, there is no reason why pressing tab should work properly until the whole of a form is loaded. It may be worth putting in a note at the top of any complicated forms, to tell your users to be patient.

Each successive tab keystroke takes the cursor to the input element with the next tab index value. After visiting the element with the highest tab index, the next tab keystroke returns the focus to the initial element. Pressing SHIFT-Tab moves the focus in reverse tabbing order.

TABINDEX values do not have to be consecutive, and they do not have to begin at 1, but they must all be different. So, a set of tab indices running 2, 4, 6, 8 is fine, as is 8, 4, 6, 2; but 2, 2, 3, 4 is forbidden as the first tab position is ambiguous.

For example:

```
<INPUT TYPE=text NAME=a TABINDEX=1>
<INPUT TYPE=button NAME=skipme NOTABSTOP>      <!--Cannot tab here-->
<INPUT TYPE=text NAME=b>                        <!--Implicit tab stop!-->
<INPUT TYPE=text NAME=c TABINDEX=2>             <!--THIRD tab stop-->
```

The NOTABSTOP attribute

NOTABSTOP simply causes elements that have it to be left out of the tab order, for instance:

```
<INPUT TYPE=text NAME=skipthisone NOTABSTOP>
```

Providing tooltip/balloon help with TITLE

Users of personal computers and Macintoshes are familiar with the little bits of text that float near the cursor when it hovers over a control: tooltips and balloon help. The tips make the systems easier to use, but do not interfere with normal operations in any way. It is proposed that HTML 4 should allow you to provide the same kind of assistance to your users. For example, if you create a set of index buttons in the form of images inside link anchors, you can add a tip to each button to make it clear what each one is supposed to do.

Tips are provided with the **TITLE** attribute. (Do not confuse this with the **TITLE** element, which gives a title to the whole document.) You can attach tips to anchors, links, images, controls, objects, applets, areas, and the new **FIELDSET** element, which is also described in this chapter. Different browsers display tips in different ways: tooltips and balloon helps are obvious possibilities, but other operating systems may display tip text in a status bar.

For example:

```
<IMG SRC=main.gif TITLE="Click where you want to go today">
<A HREF=details.html TITLE="More details of HTML 4">Details</A>
```

On Windows, tooltips are rendered with a floating yellow box, as illustrated below. Other systems use different renderings.

Details

More details of HTML 4

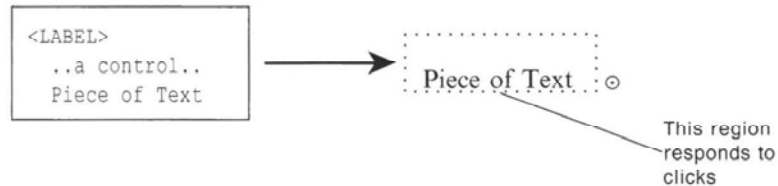
Adding clickable labels to controls

Labels for radio buttons and checkboxes were missing from earlier versions of HTML. You could put bits of text next to your form fields, but when your users clicked on the label text, nothing happened. This was a minor nuisance for most people, but a critical failing for the print-impaired. Speech-based browsers were forced into relying on **NAME** and **VALUE** attributes to access form fields.

LABEL associates a textual description with an input control, and makes it clickable. In the special case of an option button or a checkbox, a click on the text label changes the stored value (on or off). Labels can be associated either implicitly or explicitly.

Implicit labeling

Implicit labeling means wrapping the **LABEL** tag around a control such as a form field: the implication is that whatever is inside the tag is meant to be associated with the label's text. Implicit labels can contain only one object or control. The label's text may appear before or after the control. Labels cannot be nested. The diagram illustrates the mechanism.

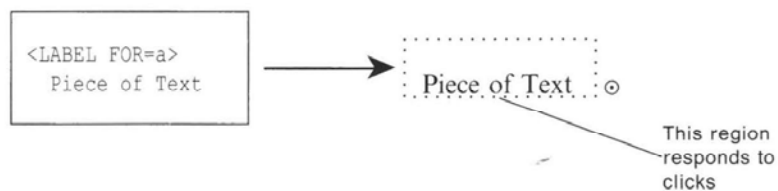


For example:

```
<LABEL ACCESSKEY=U>User Name
<INPUT TYPE="text" NAME=username ID=username>
</LABEL>
```

Explicit labeling

Sometimes, you may want to position your labels independently of their controls. Explicit labeling makes this possible. For example, you might want to have the labels in the first column of a table, and the controls in a second column. To do this, you explicitly associate each label with its control, using the **FOR** attribute. The **FOR** attribute references the control's **ID**; the control must be on the same HTML page. You can provide more than one label for a control by creating several **FOR**-marked references. The diagram illustrates the mechanism: the effect for the user is much the same as for implicit labeling.



For example:

```
<TABLE><TR>
  <TD><LABEL FOR=username ACCESSKEY=U>User Name</LABEL>
  <TD><INPUT TYPE=text NAME=username ID=username>
</TABLE>
```

Grouping fields into accessible sets

FIELDSET and **LEGEND** are meant to improve the accessibility of HTML's forms for people with disabilities. For example, in a speech-synthesized browser, they allow the blind user to move from group to group by spoken command.

FIELDSET is a nestable container (of fields and field sets). It can optionally contain a single **LEGEND** which assigns a caption to a **FIELDSET**.

LEGEND simply names the contained set of fields. If supplied, it must immediately follow the **FIELDSET** tag.

A box is drawn around each **FIELDSET** group. The group's caption can be **ALIGNED** to the **TOP** or **BOTTOM** of its fieldset if desired: the default is **TOP**. The contained fields can be organized with **<P>** or **
** into a column, or with a **<TABLE>** to achieve more elaborate effects. Such tables should be drawn without borders to avoid confusion with fieldset boxes.

The following example shows a form consisting of two banks of radio buttons. Each is constructed as a fieldset inside a table cell, complete with caption, access key, and tab index. Each individual radio button has its own access key, to be pressed once its fieldset has the focus. The HTML looks a little complicated, but once you have seen that each item is done the same way, it is quite easy to write.

```
<FORM>
<TABLE><TR>

  <TD><FIELDSET>
    <LEGEND ACCESSKEY=V TABINDEX=1>Vehicle</LEGEND>
    <P><LABEL ACCESSKEY=S>
      <INPUT TYPE=radio NAME=Vehicle VALUE=Saloon>Saloon Car</LABEL>
    <P><LABEL ACCESSKEY=E>
      <INPUT TYPE=radio NAME=Vehicle VALUE=Estate>Estate Car</LABEL>
    <P><LABEL ACCESSKEY=P>
      <INPUT TYPE=radio NAME=Vehicle VALUE=Pickup>Pickup Truck</LABEL>
    <P><LABEL ACCESSKEY=J>
      <INPUT TYPE=radio NAME=Vehicle VALUE=Jeep >Jeep</LABEL>
    </FIELDSET>
```

```

<TD><FIELDSET>
<LEGEND ACCESSKEY=E TABINDEX=2>Engine Capacity</LEGEND>
<P><LABEL ACCESSKEY=C>
<INPUT TYPE=radio NAME=Engine VALUE=S>Compact: &lt; 1000cc</LABEL>
<P><LABEL ACCESSKEY=M>
<INPUT TYPE=radio NAME=Engine VALUE=M>Medium: 1000-1999cc</LABEL>
<P><LABEL ACCESSKEY=L>
<INPUT TYPE=radio NAME=Engine VALUE=L>Large: 2000-2999cc</LABEL>
<P><LABEL ACCESSKEY=X>
<INPUT TYPE=radio NAME=Engine VALUE=X>eXtra: &gt; 3000cc</LABEL>
</FIELDSET>

</TABLE>
<P><LABEL TABINDEX=3><INPUT TYPE=submit></LABEL>
<LABEL TABINDEX=4><INPUT TYPE=reset></LABEL>
</FORM>

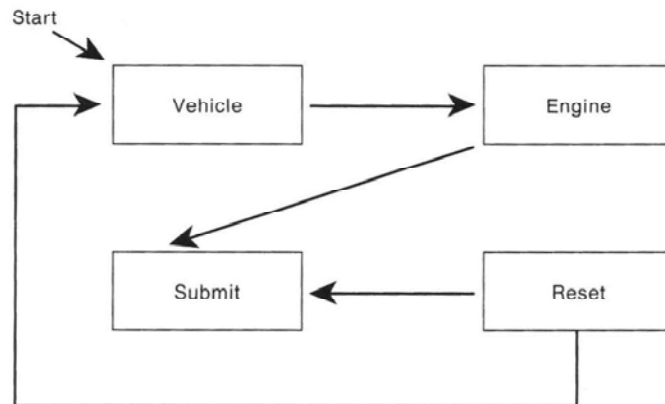
```

Here is how the example fieldset form could appear on an HTML 4 browser:

The screenshot shows a browser window titled "FieldSet" with a menu bar (File, Edit, View, Go, Bookmarks, Help). The form contains two fieldsets. The first fieldset, titled "Vehicle", has four radio buttons: "Saloon Car", "Estate Car", "Pickup Truck", and "Jeep". The second fieldset, titled "Engine Capacity", has four radio buttons: "Compact: < 1000cc", "Medium: 1000-1999cc", "Large: 2000-2999cc", and "eXtra: > 3000cc". Below these fieldsets are two buttons: "Submit Query" and "Reset". The status bar at the bottom of the browser window indicates "Document: Done".

In this example, when the form is first displayed, the Vehicle fieldset has the focus, because it has the lowest tab index (1 in this case). The user can specify an Estate Car immediately by pressing ALT-E (meaning Vehicle, Estate). Speech or other special means may activate the keys.

The Engine Capacity fieldset can then be reached by pressing the tab key. Notice that the form's Submit and Reset buttons should also be made accessible to keyboard entry: the example code provides tab indices for them. The effect of repeated tabbing in the form is therefore to rotate (forwards or backwards) around the predefined tab sequence:



In practice, we expect that the combination of HTML elements and attributes used in this example – form, table, fieldsets and captions, access keys, and tab indices – will become the normal way that forms are represented in HTML. Tools will generate all these constructs by default.

Speech-based browsers allow fieldsets to be selected directly: the user just speaks the name of the desired fieldset, as given in its caption. (Field values such as 'Engine' were already available to speech-based browsers in HTML 3.) For example, the user could say 'Engine' and then 'Medium' to locate the fieldset and push the required radio button.

11.8 Summary

This chapter illustrated the use of forms and the HTML constructs available for on-screen dialogues. These included input elements, both textual and graphical, and standard buttons to submit the form or to reset it.

Forms are not themselves difficult to write, though good visual design is a skilled art. HTML forms (running on browsers) require the support of scripts that are specifically written for the server. For more information on scripting, see Chapter 16.

HTML allows an unlimited range of sophisticated forms to be created and operated, thus – with suitable scripts and secure networks – opening up the World Wide Web (and the Internet) to business interaction and commercial traffic.